LPI exam 201 prep: System customization and automation

Intermediate Level Administration (LPIC-2) topic 213

Skill Level: Intermediate

David Mertz, Ph.D. (mertz@gnosis.cx)
Developer
Gnosis Software

Brad Huntting (huntting@glarp.com)
Mathematician
University of Colorado

01 Sep 2005

In this tutorial, David Mertz and Brad Huntting begin preparing you to take the Linux™ Professional Institute Intermediate® Level Administration (LPIC-2) Exam 201. In this seventh of eight tutorials, you learn basic approaches to scripting and automating system events, including report and status generation, clean up, and general maintenance.

Section 1. Before you start

Learn what these tutorials can teach you and how you can get the most from them.

About this series

The Linux Professional Institute (LPI) certifies Linux system administrators at junior and intermediate levels. To attain each level of certification, you must pass two LPI exams.

Each exam covers several topics, and each topic has a weight. The weights indicate the relative importance of each topic. You can expect more questions on the exam for topics with higher weight. The topics and their weights for LPI exam 201 are:

Topic 201
Linux kernel (weight 5).

Topic 202
System startup (weight 5).

Topic 203
Filesystem (weight 10).

Topic 204
Hardware (weight 8).

Topic 209
File and service sharing (weight 8).

Topic 211
System maintenance (weight 4).

Topic 213
System customization and automation (weight 3). The focus of this tutorial.

Topic 214
Troubleshooting (weight 6).

The Linux Professional Institute does not endorse any third-party exam preparation material or techniques. For details, please contact info@lpi.org.

About this tutorial

Welcome to "System customization and automation," the seventh of eight tutorials designed to prepare you for LPI exam 201. In this tutorial, you learn several basic approaches to scripting and automating system events, such as report and status generation, clean up, and general maintenance.

The tutorial is organized according to the LPI objectives for this topic:

2.213.1 Automating tasks using scripts (weight 3)
You will be able to write simple Perl scripts that make use of modules where appropriate, use the Perl taint mode to secure data, and install Perl modules from Comprehensive Perl Archive Network (CPAN). This objective includes using sed and awk in scripts, as well as using scripts to check for process execution and generating alerts by e-mail or pager when a process dies. You should be able to write and schedule automatic execution of scripts to parse logs for alerts and e-mail them to administrators, synchronize files across machines using rsync, monitor files for changes and generate e-mail alerts, and write a script that notifies administrators when specified users log in or out.

One of the task categories a system administrator must perform is to automate events that need to occur periodically and to efficiently handle other events that

occur sporadically. For automatic scheduling, your primary tools are `cron` and `at`. Tasks, whether regularly scheduled or manually launched, can be scripted with various languages, including bash, awk, Perl, or Python. Tools in the GNU text utilities are often useful as part of many processing tasks; these are most often used within bash scripts since more sophisticated languages like awk, Perl, and Python build in most of the capabilities in the text utilities.

Prerequisites

To get the most from this tutorial, you should have a basic knowledge of Linux and a working Linux system on which you can practice the commands covered in this tutorial.

Section 2. Automating periodic tasks

Configuring cron

The daemon `cron` is used to run commands periodically. You can use `cron` for a wide variety of scheduled system housekeeping and administration tasks. If there's an event or task that needs to regularly occur, it should be controlled by `cron`. `Cron` wakes up every minute to check whether it needs to do anything, but it cannot perform tasks more than once per minute. (If you need to do that, you probably want a daemon, not a "cron job.") `Cron` logs its action to the syslog facility.

`Cron` searches several places for configuration files that indicate environment settings and commands to run. The first is in /etc/crontab, which contains system tasks. The /etc/cron.d/ directory can contain multiple configuration files that are treated as supplements to /etc/crontab. Special packages can add files (matching the package name) to /etc/cron.d/, but system administrators should use /etc/crontab.

User-level `cron` configurations are stored in /var/spool/cron/crontabs/$USER. However, these should always be configured using the `crontab` tool. Using `crontab`, users can schedule their own recurrent tasks.

Scheduling daily, weekly, and monthly jobs

Jobs that should run on a simple daily, weekly, or monthly schedule -- which are the most commonly used schedules -- follow a special convention. Directories called /etc/cron.daily/, /etc/cron.weekly/, and /etc/cron.monthly/ are created to include collections of scripts to run on those respective schedules. Adding or removing scripts from these directories is a simple way to schedule system tasks. For

example, a system I maintain rotates its logs daily with a script file using:

Listing 1. Sample daily script file

```
$ cat /etc/cron.daily/logrotate
#!/bin/sh
test -x /usr/sbin/logrotate || exit 0
/usr/sbin/logrotate /etc/logrotate.conf
```

Cron and anacron

You can use `anacron` to execute commands periodically with a frequency specified in days. Unlike `cron`, `anacron` checks whether each job has been executed in the last *n* days (where *n* is the period specified for that job, as opposed to whether the current time matches the scheduled execution). If not, `anacron` runs the job's command after waiting for the number of minutes specified as the delay parameter. Therefore, on machines that are not running continuously, periodic jobs are executed once the machine is actually running (obviously, the exact timing can vary, but the task will not be forgotten).

`Anacron` reads a list of jobs from the configuration file /etc/anacrontab. Each job entry specifies a period in days, a delay in minutes, a unique job identifier, and a shell command. For example, on one Linux system I maintain, `anacron` is used to run daily, weekly, and monthly jobs even if the machine is not running at the scheduled time of day:

Listing 2. Sample anacron configuration file

```
$ cat /etc/anacrontab
# /etc/anacrontab: configuration file for anacron
SHELL=/bin/sh
PATH=/sbin:/bin:/usr/sbin:/usr/bin
# These replace cron's entries
1         5  cron.daily    nice run-parts --report /etc/cron.daily
7        10  cron.weekly   nice run-parts --report /etc/cron.weekly
@monthly 15  cron.monthly  nice run-parts --report /etc/cron.monthly
```

The contents of a crontab

The format of /etc/crontab (or the contents of /etc/cron.d/ files) is slightly different from that of user crontab files. Basically, this just amounts to an extra field in /etc/crontab that indicates the user a command runs as. This is not needed for user crontab files since they are already stored in a file matching username (/var/spool/cron/crontabs/$USER).

Each line of /etc/crontab either sets an environment variable or configures a recurring job. Comment and blank lines are ignored. For cron jobs, the first five fields specify times to run (where each zero-based field may have a list and/or a range). The fields are minute, hour, day of month, month, day of week (space- or

tab-separated). An asterisk (*) in any position indicates *any*. For example, to run a task at midnight on Tuesdays and Thursdays during August through October, you could use:

```
# line in /etc/crontab
0 0 * 7-9 2,5 root /usr/local/bin/the-task -opt1 -opt2
```

Using special scheduling values

Some common scheduling patterns have shortcut names you can use in place of the first five fields:

@reboot
Run once, at startup.

@yearly
Run once a year, "0 0 1 1 *".

@annually
Same as @yearly.

@monthly
Run once a month, "0 0 1 * *".

@weekly
Run once a week, "0 0 * * 0".

@daily
Run once a day, "0 0 * * *".

@midnight
Same as @daily.

@hourly
Run once an hour, "0 * * * *".

For example, you could have a configuration containing:

```
@hourly root /usr/local/bin/hourly-task
0,29 * * * * root /usr/local/bin/twice-hourly-task
```

Using crontab

To set up a user-level scheduled task, use the `crontab` command (as opposed to the /etc/crontab file). Specifically, `crontab -e` launches an editor to modify a file. You can list current jobs with `crontab -l` and remove the file with `crontab -r`. Or you can specify `crontab -u user` to schedule tasks for a given user, but the default is to do so for yourself (permission limits apply).

The /etc/cron.allow file, if present, must contain the names of all users allowed to schedule jobs. Alternately, if there is no /etc/cron.allow, then a user must not be in the /etc/cron.deny file if allowed to schedule tasks. If neither file exists, everyone can use `crontab`.

Section 3. Automating one-time tasks

Using the at command

If you need to schedule a task to run in the future, you can use the `at` command, which takes a command from STDIN or from a file (using the `-f` option), and accepts time descriptions in a flexible collection of formats.

A family of commands is used in association with the `at` command: `atq` lists pending tasks; `atrm` removes a task from the pending queue; and `batch` works much like `at`, except it defers running a job until the system load is low.

Permissions

Similar to /etc/cron.allow and /etc/cron.deny, the `at` command has /etc/at.allow and /etc/at.deny files to configure permissions. The /etc/at.allow file, if present, must contain all users allowed to schedule jobs. Alternately, if there is no /etc/at.allow, then a user must not be in /etc/at.deny if allowed to schedule tasks. If neither file exists, everyone may use `at`.

Time specifications

See the manpage on your `at` version for full details. You can specify a particular time as `HH:MM`, which schedules an event to happen when that time next occurs. (If the time has already passed today, it means tomorrow.) If you use 12-hour time, you can also add a.m. or p.m. You can give a date as `MMDDYY`, `MM/DD/YY`, `DD.MM.YY`, or `month-name-day`. You can also increment from the current time with `now + N units`, in which *N* is a number and *units* are minutes, hours, days, or weeks. The words *today* and *tomorrow* keep their obvious meaning, as do *midnight* and *noon* (*teatime* is 4 p.m.). Some examples:

```
% at -f ./foo.sh 10am Jul 31 % echo 'bar -opt' | at 1:30
tomorrow
```

The exact definition of the time specification is in /usr/share/doc/at/timespec.

Section 4. Tips for scripts

Outside resources

Many excellent books are available on awk, Perl, bash, and Python. The coauthor of this tutorial (naturally) recommends his own title, *Text Processing in Python* , as a good starting point for scripting in Python.

Most scripts you write for system administration focus on text manipulation such as extracting values from logs and configuration files and generating reports and summaries. It also means cleaning up system cruft and sending notifications of tasks performed.

The most common scripts in Linux system administration are written in bash. bash itself has relatively few built-in capabilities, but bash makes it particularly easy to utilize external tools (including basic file utilities such as ls, find, rm, and cd) and text tools (like those found in the GNU text utilities).

Bash tips

One particularly helpful setting to include in bash scripts that run on a schedule is the `set -x` switch, which echoes the commands run to STDERR. This is helpful in debugging scripts when they don't produce the desired effect. Another useful option during testing is `set -n`, which causes a script to look for syntax problems, but not actually to run. Obviously, you don't want a `-n` version scheduled in `cron` or `at`, but to get it up and running, it can help.

Listing 3. Sample cron job that runs a bash script

```
#!/bin/bash
exec 2>/tmp/my_stderr
set -x
# functional commands here
```

This redirects STDERR to a file and outputs the commands run to STDERR. Examining that file later can be useful.

The manpage for bash is good, though quite long. You may find all the options that the built-in `set` can accept particularly interesting.

A common task in a system administration script is to process a collection of files, often with the files of interest identified using the `find` command. However, a problem can arise when file names contain white space or newline characters. Much of the looping and processing of file names you are likely to do can be confused by these internal white space characters. For example, these two commands are

different:

```
% rm foo bar baz bam
% rm 'foo bar' 'baz bam'
```

The first command unlinks four files (assuming they exist to start with); the second removes just two files, each with an internal space in the name. File names with spaces are particularly common in multimedia content.

Fortunately, the GNU version of the `find` command has a `-print0` option to NULL terminate each result; and the `xargs` command has a corresponding `-0` command to treat arguments as NULL separated. Putting these together, you can clean up stray files that might contain white space in their names using:

Listing 4. Cleaning up file names with spaces

```
#!/bin/bash
# Cleanup some old files
set -x
find /home/dqm \( -name '*.core' -o -name '#*' \) -print0 \
 | xargs -0 rm -f
```

Perl taint mode

Perl has a handy switch `-T` to enable taint mode. In this mode, Perl takes a variety of extra security precautions, but primarily it limits execution of commands arising from external input. If you use `sudo` execution, taint mode might be enabled automatically, but the safest thing is to start your administration scripts with:

```
#!/usr/local/bin/perl -T
```

Once you do this, all command line arguments, environment variables, locale information (see perllocale), results of certain system calls (`readdir()`, `readlink()`, the variable of `shmread()`, the messages returned by `msgrcv()`, the password, gcos and shell fields returned by the `getpwxxx()` calls), and all file inputs are marked as "tainted." Tainted data cannot be used directly or indirectly in any command that invokes a sub-shell nor in any command that modifies files, directories, or processes, with a few exceptions.

It's possible to untaint particular external values by carefully checking them for expected patterns:

Listing 5. Untainting external values

```
if ($data =~ /^([-\@\w.]+)$/) {
   $data = $1;                     # $data now untainted
} else {
   die "Bad data in $data";      # log this somewhere
}
```

Perl CPAN packages

One of the handy things about Perl is that it comes with a convenient mechanism for installing extra support packages; it's called Comprehensive Perl Archive Network (CPAN). RubyGems is similar in function. Python, unfortunately, does not yet have an automated installation mechanism, but it comes with more in the default installation. Simpler languages like bash and awk do not really have many add-ons to install in an analogous sense.

The manpage on the `cpan` command is a good place to get started, especially if you have a task to perform for which you think someone might have done most of the work already. Look for candidate modules at CPAN.

`cpan` has both an interactive shell and a command-line operation. Once configured (run the interactive shell once to be prompted for configuration options), `cpan` handles dependencies and download locations in an automated manner. For example, suppose you discover you have a system administration task that involves processing configuration files in YAML (yaml Ain't Markup Language) format. Installing support for YAML is as simple as:

```
% cpan -i YAML # maybe with 'sudo' first
```

Once installed, your scripts can contain `use YAML;` at the top. This goes for any capabilities for which someone has created a package.

Resources

Learn

- At the LPIC Program, find task lists, sample questions, and detailed objectives for the three levels of the Linux Professional Institute's Linux system administration certification.
- "Understanding Linux configuration files" (developerWorks, December 2001) shows you how to set up configuration files on a Linux system that control user permissions, system applications, daemons, services, and other administrative tasks in a multiuser, multitasking environment.
- The developerWorks tutorial "Using the GNU text utilities" (developerWorks, March 2004) introduces you to scripting utilities.
- These scripting articles on developerWorks provide lots of resources on using scripts to automate tasks in Linux.
- *Text Processing in Python* by David Mertz, co-author of this tutorial, is an excellent source for Python scripts.
- Find more resources for Linux developers in the developerWorks Linux zone.

Get products and technologies

- The CPAN Module Archive is your source for Perl modules.
- Order the SEK for Linux: Get this two-DVD set containing the latest IBM trial software for Linux from DB2®, Lotus®, Rational®, Tivoli®, and WebSphere®.
- Build your next development project on Linux with IBM trial software, available for download directly from developerWorks.

Discuss

- Participate in the discussion forum for this content.
- Get involved in the developerWorks community by participating in developerWorks blogs.

About the authors

David Mertz, Ph.D.
David Mertz is Turing complete, but probably would not pass the Turing Test. For more about his life, see his personal Web page. He's been writing the developerWorks columns *Charming Python* and *XML Matters* since 2000. Check out his book *Text Processing in Python* . You can contact David at mertz@gnosis.cx.

Brad Huntting
Brad has been doing UNIX® systems administration and network engineering for about 14 years at several companies. He is currently working on a Ph.D. in Applied Mathematics at the University of Colorado in Boulder, and pays the bills by doing UNIX support for the Computer Science department. Contact Brad at huntting@glarp.com.